Sparse Suffix and LCP Array: Simple, Direct, Small, and Fast

Lorraine A. K. Ayad[1], Grigorios Loukides[2], Solon P. Pissis[3,4], **Hilde Verbeek**[3]

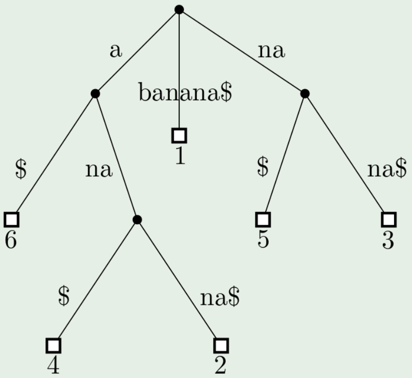
[1]Brunel University London, UK
[2]King's College London, UK
[3]CWI, Amsterdam, Netherlands
[4]Vrije Universiteit, Amsterdam, Netherlands

Dutch Optimization Seminar, 7 December 2023

Suffix trees

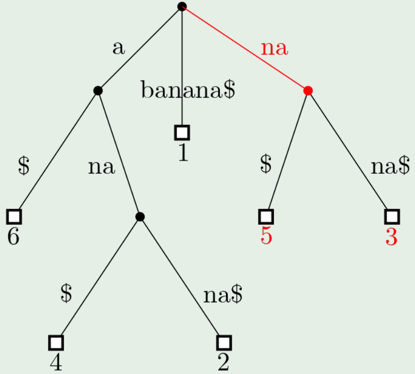
- Indexing large amounts of text or DNA requires small data structures and fast algorithms
- Suffix tree: trie of all suffixes of a string

Example (Suffix tree of "banana")



Suffix trees

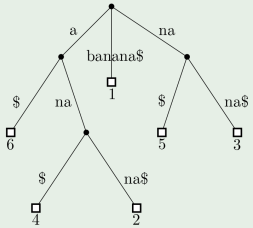
Example (Finding all occurrences of "na" in "banana")



Suffix array and LCP array

- Suffix array: all suffixes of the string sorted lexicographically
- LCP array: longest common prefix of two consecutive suffixes
- Correspondence with suffix tree
- Takes less space in practice

Example (Suffix tree, suffix array and LCP array of "banana")



i	suffix	SA[i]	LCP[i]
1	a	6	0
2	ana	4	1
3	anana	2	3
4	banana	1	0
5	na	5	0
6	nana	3	2

Sparse suffix and LCP array

- Let B be a set of positions in the string T
- Sparse suffix array: suffixes starting at positions in B, sorted
- Sparse LCP array: longest common prefixes of SSA

Example (Sparse suffix and LCP array of "abracadabra")

Let T = abracadabra and $B = \{1, 5, 6, 8\}$. The relevant suffixes are abracadabra, cadabra, adabra, abra. Sorting these gives:

i	suffix	SSA[i]	SLCP[i]
1	abra	8	0
2	abracadabra	1	4
3	adabra	6	1
4	cadabra	5	0

Sparse Suffix Sorting

Sparse Suffix Sorting

Given: string $T \in \Sigma^n$, set B of b indices in $[1, n]$
Asked: the arrays SSA and SLCP

- Building the full suffix and LCP array takes too much space
- Can we design an algorithm
  - in (near-)linear time,
  - using $\mathcal{O}(b)$ space,
  - that constructs SSA and SLCP more or less directly,
  - and is simple to understand and implement?

Sparse Suffix Sorting

Time	Space	Notes
Kärkkäinen, Sanders, and Burkhardt 2006		
$\mathcal{O}(n^2/s)$	$\mathcal{O}(s)$	for $s \in [b, n]$
Bille et al. 2016		
$\mathcal{O}(n \log^2 b)$	$\mathcal{O}(b)$	Monte Carlo
$\mathcal{O}(n \log^2 n + b^2 \log b)$	$\mathcal{O}(b)$	Las Vegas
I, Kärkkäinen, and Kempa 2014		
$\mathcal{O}(n + (bn/s) \log s)$	$\mathcal{O}(b)$	Monte Carlo
$\mathcal{O}(n \log b)$	$\mathcal{O}(b)$	Las Vegas
Gawrychowski and Kociumaka 2017		
$\mathcal{O}(n)$	$\mathcal{O}(b)$	Monte Carlo
$\mathcal{O}(n\sqrt{\log b})$	$\mathcal{O}(b)$	Las Vegas
Birenzwige, Golan, and Porat 2020		
$\mathcal{O}(n)$	$\mathcal{O}(b)$	Las Vegas
$\mathcal{O}(n \log \frac{n}{b})$	$\mathcal{O}(b)$	$b = \Omega(\log n)$
Fischer, I, and Köppl 2020		
$\mathcal{O}(c\sqrt{\log n} + b \log b \log n \log^* n)$	$\mathcal{O}(b)$	"Restore" model
Prezza 2021		
$\mathcal{O}(n + b \log^2 n)$	$\mathcal{O}(1)$	Restore model, Monte Carlo

Table: Existing algorithms for Sparse Suffix Sorting

Sparse Suffix Sorting

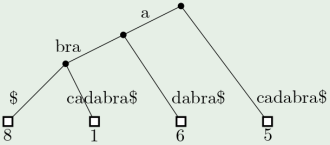
Our contributions:

- an $\mathcal{O}(n \log b)$ time algorithm that uses $8b + o(b)$ machine words of space
- an improved version, that runs in $\mathcal{O}(n)$ time if the number of suffixes with long LCPs is sufficiently small
- experimental results supporting the time and space complexity

Overview

- Based on work by I et al.[1]
- Simulate the sparse suffix tree, then extract SSA and SLCP from that
- Our contribution: implement using an array-based approach rather than a tree, which saves time and space in practice

Example (Sparse suffix tree, sparse suffix array and LCP array)



i	suffix	SSA[i]	SLCP[i]
1	abra	8	0
2	abracadabra	1	4
3	adabra	6	1
4	cadabra	5	0

[1] I, Kärkkäinen, and Kempa 2014

Overview

1. Iteratively create the hierarchy of LCP groups
2. Sort the entries of each LCP group
3. Build SSA and SLCP based on the LCP groups

Definition (LCP group)

An LCP group is a triple $(id, \{b_1, \ldots, b_k\}, lcp)$ where

- id is its unique identifier
- $b_1, \ldots, b_k$ are each either an entry from B (indicating a suffix) or another LCP group
- all suffixes in the group have a common prefix of at least lcp characters

Step 1: building LCP groups

$T = \texttt{c a t e r p i l l a r c a p i l l a r y \$}$ $(n = 20)$

$\texttt{c}$ 1, $\texttt{a}$ 2, $\texttt{p}$ 3, $\texttt{a}$ 4, $\texttt{p}$ 5, $\texttt{a}$ 6

$\boxed{7, \{1, 2, 3, 4, 5, 6\}, 0}$

Start with one group having an LCP value of 0. We will refine the groups for decreasing powers of 2, starting at 16.
If some suffixes have a common prefix, they will be put together into a new group.

We check for matches using Karp-Rabin fingerprints and a hash table.

Step 1: building LCP groups

$T = \texttt{c a t e r p i l l a r c a p i l l a r y \$}$ ($n = 20$)
1 2 3 4 5 6

$\boxed{7, \{1, 2, 3, 4, 5, 6\}, 0}$

Prefixes of length 16:

1: `caterpillarcapil`
2: `aterpillarcapill`
3: `pillarcapillary$`
4: `arcapillary$`
5: `pillary$`
6: `ary$`

(no match)

Step 1: building LCP groups

$T = \texttt{c a t e r p i l l a r c a p i l l a r y \$}$ $(n = 20)$

1 2 3 4 5 6

$\boxed{7, \{1, 2, 3, 4, 5, 6\}, 0}$

Prefixes of length 8:

1: `caterpil`
2: `aterpill`
3: `pillarca`
4: `arcapill`
5: `pillary$`
6: `ary$`

(still no match)

Step 1: building LCP groups

$T = \texttt{c a t e r p i l l a r c a p i l l a r y \$}$ $(n = 20)$

1 2 3 4 5 6

$\boxed{7, \{1, 2, 3, 4, 5, 6\}, 0}$

Prefixes of length 4:

1: `cate`
2: `ater`
3: `pill`
4: `arca`
5: `pill`
6: `ary$`

Suffixes 3 and 5 have a common prefix of length 4.

Step 1: building LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 3 (8) 4 5 6

$\boxed{7, \{1, 2, 4, 6, 8\}, 0}$ $\boxed{8, \{3, 5\}, 4}$

Prefixes of length 4:

1: `cate`
2: `ater`
3: `pill`
4: `arca`
5: `pill`
6: `ary$`

Create a new group for suffixes 3 and 5.

Step 1: building LCP groups

$T = $ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 3 (8) 4 5 6

$\boxed{7, \{1, 2, 4, 6, 8\}, 0}$ $\boxed{8, \{3, 5\}, 4}$

Extend prefixes by 2:

1: `ca`	3: `(pill)ar`
2: `at`	5: `(pill)ar`
4: `ar`	
6: `ar`	
8: `pi` (*)	

Suffixes 4 and 6 in group 7 have a common prefix of length 2, and suffixes 3 and 5 in group 8 have a common prefix of length $4 + 2$.

Step 1: building LCP groups

$T =$ c a t e r p i l l a r c a p i l l a r y \$ ($n = 20$)

1 2 3 (8) 4 (9) 5 6

$\boxed{7, \{1, 2, 8, 9\}, 0}$ $\boxed{8, \{3, 5\}, 4}$ $\boxed{9, \{4, 6\}, 2}$

Extend prefixes by 2:

1: ca
2: at
4: ar
6: ar
8: pi (*)

3: (pill)ar
5: (pill)ar

Create a new group for suffixes 4 and 6.

Step 1: building LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 3 (8) 4 (9) 5 6

$\boxed{7, \{1, 2, 8, 9\}, 0}$ $\boxed{8, \{3, 5\}, 6}$ $\boxed{9, \{4, 6\}, 2}$

Extend prefixes by 2:

1: `ca`
2: `at`
4: `ar`
6: `ar`
8: `pi` (*)

3: `(pill)ar`
5: `(pill)ar`

Update the LCP value for group 8.

Step 1: building LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 3 (8) 4 (9) 5 6

$\boxed{7, \{1, 2, 8, 9\}, 0}$ $\boxed{8, \{3, 5\}, 6}$ $\boxed{9, \{4, 6\}, 2}$

Extend prefixes by 1:

1: `c` 3: `(pillar)c` 4: `(ar)c`
2: `a` 5: `(pillar)y` 6: `(ar)y`
8: `p` (*)
9: `a` (*)

Suffix 2 and group 9 in group 7 have a common prefix of length 1.

Step 1: building LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 (10), 3 (8), 4 (9), 5, 6

$7, \{1, 8, 10\}, 0$	$8, \{3, 5\}, 6$	$9, \{4, 6\}, 2$	$10, \{2, 9\}, 1$

Extend prefixes by 1:

1: `c`
2: `a`
8: `p` (*)
9: `a` (*)

3: `(pillar)c`
5: `(pillar)y`

4: `(ar)c`
6: `(ar)y`

Create a new group for 2 and 9.

Step 1: building LCP groups

$T = $ c a t e r p i l l a r c a p i l l a r y \$ $(n = 20)$

c	a	t	e	r	p	i	l	l	a	r	c	a	p	i	l	l	a	r	y	\$
1	2 (10)				3 (8)				4 (9)				5				6			

$\boxed{7, \{1, 8, 10\}, 0}$ $\boxed{8, \{3, 5\}, 6}$ $\boxed{9, \{4, 6\}, 2}$ $\boxed{10, \{2, 9\}, 1}$

Now all the LCP values are correct, and step 1 is finished.

Step 2: sorting the LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 (10) 3 (8) 4 (9) 5 6

$\boxed{7, \{1, 8, 10\}, 0}$ $\boxed{8, \{3, 5\}, 6}$ $\boxed{9, \{4, 6\}, 2}$ $\boxed{10, \{2, 9\}, 1}$

1: `c`	3: `(pillar)c`	4: `(ar)c`	2: `(a)t`
8: `p`	5: `(pillar)y`	6: `(ar)y`	9: `(a)r`
10: `a`			

We already have all the LCP values, so we can compare suffixes by just looking at the character after the LCP.

Step 2: sorting the LCP groups

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 (10) 3 (8) 4 (9) 5 6

$\boxed{7, \{10, 1, 8\}, 0}$ $\boxed{8, \{3, 5\}, 6}$ $\boxed{9, \{4, 6\}, 2}$ $\boxed{10, \{9, 2\}, 1}$

1: `c`	3: `(pillar)c`	4: `(ar)c`	2: `(a)t`
8: `p`	5: `(pillar)y`	6: `(ar)y`	9: `(a)r`
10: `a`			

Sort each LCP group using e.g. in-place MergeSort.

Step 3: building the SSA and SLCP

$T =$ `c a t e r p i l l a r c a p i l l a r y $` $(n = 20)$

1 2 (10) 3 (8) 4 (9) 5 6

$7, \{10, 1, 8\}, 0$ | $8, \{3, 5\}, 6$ | $9, \{4, 6\}, 2$ | $10, \{9, 2\}, 1$

Build SSA and SLCP using a depth-first search on the LCP group hierarchy. The LCP value of two suffixes is that of their "lowest common ancestor" group.

i	suffix	SSA[i]	SLCP[i]
1	arcapillary	4	0
2	ary	6	2
3	aterpillarcapillary	2	1
4	caterpillarcapillary	1	0
5	pillarcapillary	3	0
6	pillary	5	6

Karp-Rabin fingerprints

Lemma (I, Kärkkäinen, and Kempa 2014)

Given a string T of length n and an integer s, we can create a data structure of size $\mathcal{O}(s)$ in $\mathcal{O}(n)$ time that allows us to find the KR-fingerprint of any length-k substring of T, in $\mathcal{O}(\min\{k, n/s\})$ time.

Complexity

- Pre-processing: $\mathcal{O}(n)$ time
- Step 1: $\mathcal{O}((bn/s)\log s)$ time
  - $\mathcal{O}(\log n)$ rounds, $\mathcal{O}(b)$ fingerprints each round
  - Long fingerprints (first $\log s$ rounds): $\mathcal{O}((bn/s)\log s)$
  - Short fingerprints (last $\log n - \log s$): amortized $\mathcal{O}(bn/s)$
- Step 2: $\mathcal{O}(n)$ time
  - Sorting $\mathcal{O}(b)$ items over at most b groups
  - b is low: merge sort; b is high: radix sort
  - Either case, $\mathcal{O}(n)$ time
- Step 3: $\mathcal{O}(b)$ time
  - DFS over the $\mathcal{O}(b)$ groups and suffixes: $\mathcal{O}(b)$ time

Complexity

Theorem

Given $T \in \Sigma^n$, set B of b indices in $[1, n]$ and an integer $s \in [b, n]$, SSA and SLCP can be computed in $\mathcal{O}(n + (bn/s) \log s)$ time using $s + 7b + o(b)$ machine words of space.

- If $s = b$, then $\mathcal{O}(n \log b)$ time and $8b + o(b)$ space
- Implementing the LCP groups sequentially instead of as a tree improves running time in practice
- Karp-Rabin fingerprints are randomized; the output is correct with high probability

Parameterized algorithm

- Most suffixes will likely have short LCPs
- Save time by starting at lower powers of 2
  - Recall, substrings shorter than n/s can be fingerprinted faster
  - Some LCP values may be underestimated
- We can easily identify the “incorrect” LCP values by looking at the next character
- All other suffixes are already at the right position in SSA

Parameterized algorithm

1. Run the algorithm, starting at $2^{\lfloor \log \frac{n}{b} \rfloor}$ (and $s = b$)
   - Longest LCP that can be found is $\ell = 2^{\lfloor \log \frac{n}{b} \rfloor + 1} - 1$
2. Identify suffixes that have LCP value ℓ and have the $\ell + 1$-th character in common with their neighbor in SSA
3. Run the algorithm again with all powers of 2, just on the identified suffixes
4. Insert results of the second run in the same positions in SSA and SLCP

Example

Step 1: Sort up to $\ell = 7$ positions in the first round.

Step 1	LCP*
	0
gratuitous	
	0
harbingers	
	4
harborserv	
	7
harborseal	
	1
howevertha	
	1
hungrycate	
	0
integratio	
	7
integratin	
	7
integrated	
	0
omniscient	

Example

Step 2: Identify suffixes with LCP longer than ℓ.

Step 1	LCP*	Step 2
gratuitous	0	
harbingers	0	
harborserv	4	harborserv
harborseal	7	harborseal
howevertha	1	
hungrycate	1	
integratio	0	integratio
integratin	7	integratin
integrated	7	integrated
omniscient	0	

Example

Step 3: Re-run the algorithm on just these suffixes.

Step 1	LCP*	**Step 2**	**Step 3**	LCP
	0			
gratuitous				
	0			
harbingers				
	4			0
harborserv		harborserv	harborseal	
	7			8
harborseal		harborseal	harborserv	
	1			
howevertha				
	1			
hungrycate				
	0			0
integratio		integratio	integrated	
	7			8
integratin		integratin	integratin	
	7			9
integrated		integrated	integratio	
	0			
omniscient				

Example

Step 4: Insert re-sorted suffixes in the same positions.

Step 1	LCP*	**Step 2**	**Step 3**	LCP	**Step 4**	LCP
gratuitous	0				gratuitous	0
harbingers	0				harbingers	0
harborserv	4	harborserv	harborseal	0	harborseal	4
harborseal	7	harborseal	harborserv	8	harborserv	8
howevertha	1				howevertha	1
hungrycate	1				hungrycate	1
integratio	0	integratio	integrated	0	integrated	0
integratin	7	integratin	integratin	8	integratin	8
integrated	7	integrated	integratio	9	integratio	9
omniscient	0				omniscient	0

Complexity

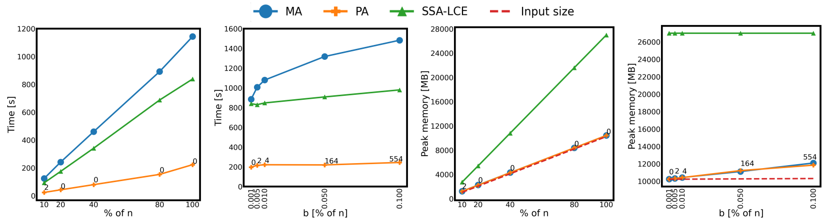
- Let b' be the number of incorrectly sorted suffixes
- First round: $\mathcal{O}(n)$ (shorter fingerprints)
- Second round: $\mathcal{O}(n + (b'n/b)\log b)$ (fewer suffixes)
- Other steps: $\mathcal{O}(b)$

Theorem

If b' of the suffixes have an associated LCP longer than ℓ, SSA and SLCP can be computed in $\mathcal{O}(n + (b'n/b)\log b)$ time using $8b + 4b' + o(b)$ machine words of space.

- If $b' = \mathcal{O}(b/\log b)$, this runs in $\mathcal{O}(n)$ time
- In practice, b' is often extremely small

Experimental results



(a) Time for varying n

(b) Time for varying b

(c) Memory for varying n

(d) Memory for varying b

Figure: Results on 10.23 GB of Amazon reviews, compared to a benchmark algorithm SSA-LCE (Prezza 2021). The values of b' are shown on top of the data points of PA.

Thank you!

Paper on arXiv: `https://arxiv.org/abs/2310.09023`

References

- Bille, Philip et al. (2016). "Sparse Text Indexing in Small Space". In: *ACM Trans. Algorithms.*
- Birenzwige, Or, Shay Golan, and Ely Porat (2020). "Locally Consistent Parsing for Text Indexing in Small Space". In: *SODA 2020.*
- Fischer, Johannes, Tomohiro I, and Dominik Köppl (2020). "Deterministic Sparse Suffix Sorting in the Restore Model". In: *ACM Trans. Algorithms.*
- Gawrychowski, Pawel and Tomasz Kociumaka (2017). "Sparse Suffix Tree Construction in Optimal Time and Space". In: *SODA 2017.*
- I, Tomohiro, Juha Kärkkäinen, and Dominik Kempa (2014). "Faster Sparse Suffix Sorting". In: *STACS 2014.*
- Kärkkäinen, Juha, Peter Sanders, and Stefan Burkhardt (2006). "Linear work suffix array construction". In: *J. ACM.*
- Prezza, Nicola (2021). "Optimal Substring Equality Queries with Applications to Sparse Text Indexing". In: *ACM Trans. Algorithms.*